Constructive Negation of Arithmetic Constraints Using Dataflow Graphs

JOHN G. CLEARY
Department of Computer Science, University of Waikato, Hamilton, New Zealand
jcleary@cs.waikato.ac.nz

Abstract. A system which extracts a dataflow graph from sets of arithmetic constraints is described. This information is used to simplify constraints and to extract positive information from negations of constraints. The context for this work is a Prolog implementation where intervals are used to represent the underlying arithmetic variables. The system uses simple information about the existence of solutions of primitive constraints to derive the dataflow graph. This makes the system easily extensible to new primitives and domains. A practical implementation over both real and integer arithmetic is described and an extended example of its application given.

Keywords: constructive negation, logic programming, interval arithmetic, constraint logic programming

1. Introduction

A number of different execution mechanisms have been introduced in recent years under the title of constraint logic programming. Such systems allow the result of a logic program to be some logical formula which “constrains” the set of answers. In general such constraints include arbitrary formulas including negations and existential quantifiers. The problem is to ensure that these answers are as informative as possible, that they give as much positive information about the results as possible. That is, the negations and quantifiers in the answers should be eliminated where possible. There are a number of situations where negations might arise.

For example, Chan (1988) defines a “constructive negation” procedure over constraints involving equality and inequality. He shows a complete quantifier elimination procedure which allows any answer to be reduced to a disjunction of conjunctions of primitive constraints where each primitive constraint is either of the form \( X = Y \) or \( \forall Z X \neq Y \). He describes an \( SLD-CN\) execution mechanism which implements a negated goal by executing it and collecting all its solutions and then rewriting this set of disjuncts. Chan’s work has been expanded and refined in Stuckey (1991).

Constructive negation might be applied in other circumstances. At compile time it might be used to simplify negations appearing in the source thus reducing the work required at run time. Two major benefits can be expected from using constructive negation. One is the reduction of the complexity of the final answers returned at execution time. The other is that by returning positive information about constraints it may be possible to interact with other constraints in the system and achieve a more direct and faster convergence to a final solution. Examples of both these phenomena are given later.
1.1. Arithmetic

One application of constraint logic programming is arithmetic. As described in Cleary (1987) this involves the expression of arithmetic statements as a set of constraints on the arithmetic variables. For example, a statement such as \( Z = 2 \times X + Y \) would be rendered as, \( \text{add}(Y, T, Z), \text{mult}(2, X, T) \): where add and mult are primitive constraints for addition and multiplication. As well as using constraints to express the arithmetic relationships, the values of the variables are also expressed as intervals. For example, that \( X \) lies between 1 and 2.5 and \( Y \) between 0 and 0.5, is expressed by the constraint \( 1 \leq X \leq 2.5, 0 \leq Y \leq 0.5 \); which for brevity will be written as \( 1 \leq X \leq 2.5, 0 \leq Y \leq 0.5 \). Sometimes, it is possible to deduce new positive information from such a set of constraints. For example, combining the interval constraints on \( X \) and \( Y \) with \( Z = 2 \times X + Y \) it can be deduced that \( 2.0 \leq Z \leq 5.5 \). Doing arithmetic this way provides a powerful problem solving capability. For example an equation such as, \( X = Y^2 \), can be used to compute values for both \( X \) and \( Y \) given the other value.

A constraint logic programming based system such as this requires a sophisticated run-time structure. When a constraint is first executed it checks all its variables and further constrains their intervals if possible. Consider a set of variables which have been given the following interval constraints:

\[
0 \leq X \leq 4, 1 \leq Y \leq 4, 7 \leq Z \leq 9.
\]

If now the constraint \( \text{add}(X, Y, Z) \) is executed it *relaxes* these constraints and deduces the largest intervals which could possibly satisfy both the original interval constraints and the new add constraint. In this case the process of relaxation leads to the new set of interval constraints:

\[
3 \leq X \leq 4, 3 \leq Y \leq 4, 7 \leq Z \leq 8
\]

Note that all three of the intervals have been narrowed. At this point it is necessary to retain the add constraint in the system as it may be needed later for further relaxation. A constraint that is being held in this way will be said to be *delayed*. It will remain delayed until one of its variables is altered. For example if the constraint \( Y \leq 3.5 \) is now executed, the add constraint will be woken and further relaxation will give:

\[
3.5 \leq X \leq 4, 3 \leq Y \leq 3.5, 7 \leq Z \leq 7.5
\]

(again all three intervals have been narrowed). If there are other delayed constraints on \( X \), \( Y \) and \( Z \) they also will be woken at this point. The add constraint must still remain delayed. At some point the variables may be given exact values, for example, \( X = 3.5 \). This will cause the add constraint to be woken and to relax the variables which gives:

\[
X = 3.5, Y = 3.5, Z = 7
\]

Because all its variables now have point values the add constraint will never need to be woken again, so it *succeeds* and is removed from the system. A constraint is said to be